UML-based Alias Control

Yin Liu
Department of Computer Science
Rensselaer Polytechnic Institute

liuy@cs.rpi.edu

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute

milanova@cs.rpi.edu

ABSTRACT

We propose a mechanism for alias control which is based on the Unified Modeling Language (UML). Specifically, we propose use of *ownership* and *immutability* constraints on UML associations and verification of these constraints through reverse engineering. These constraints inherently support software design principles, and impose requirements on the implementation that may help prevent serious program flaws.

We propose implementation-level models for ownership and immutability that capture well the meaning of these concepts in modeling, and we develop novel static ownership and immutability inference analyses. We perform an empirical investigation on several relatively large Java programs. The results indicate that the inference analyses are precise and practical. Therefore, the analyses can be integrated in reverse engineering tools and can help support effective reasoning about software quality and software security.

1. INTRODUCTION

Unexpected aliasing between objects can seriously compromise software quality and software security. For example, in Java 1.1 the security function `Class.getSigners` mistakenly returned a reference to an internal array of signers; untrusted clients could modify this array and compromise the security of the system. Current languages such as Java do not provide effective mechanisms for preventing unexpected aliasing. Therefore, it is important to develop such mechanisms and advance their usage in software practice.

This paper proposes use of alias-control constraints on Unified Modeling Language (UML) [42] class diagrams, and verification of these constraints through reverse engineering. UML class diagrams describe the architecture of the program in terms of classes and *associations* that model interclass relationships; they are scalable and informative models, widely-used in software engineering practice.

Specifically, we propose use of *ownership* and *immutability* constraints on UML associations. An association from class A to class B marked as `owned` at design level, states a requirement for ownership and no *representation exposure* at implementation level. Intuitively, each A object must control the B objects it references through this association. An association from class A to class B marked as `read-only` states a requirement for immutability at the implementation level. An A object cannot modify the heap structure rooted at the B object it references through this association.

Ownership and immutability constraints on UML associations inherently support software design principles such as "Low Coupling" and "Information Expert" [23]. Most importantly, the constraints force reasoning about alias control at design level and impose requirements on the implementation. These requirements can be continuously verified through reverse engineering which may prevent serious flaws due to representation exposure such as the `Signers` bug.

The goals of this work are (i) to define implementation-level ownership and immutability models that capture the meaning of these notions at design level, and (ii) to develop practical and precise analyses that infer ownership and immutability in accordance with these models. The definition of implementation-level ownership is based on *owners-as-dominators* [13, 35]; in this model the owner object should dominate an owned object—that is, all access paths to the owned object should pass through its owner. As pointed out by Clarke et al. [13], the owners-as-dominators model captures well the notion of *composition* in modeling [42]; thus, design-level ownership constraints generalize composition relationships. The definition of immutability requires that an enclosing object have read-only access to an enclosed immutable object—that is, the methods invoked on the enclosing object cannot change (directly, or through callees) the heap structure rooted at the immutable object.

We propose two novel static analyses for Java, one for ownership inference and one for immutability inference; the analyses work directly on Java code and do not require annotations by the programmer. Consider a reverse engineered association from class A to class B. If the ownership inference determines that all A objects own the corresponding B objects referenced through this association, the analysis marks the association as `owned`. Analogously, if the immutability inference determines that all A objects never modify the B objects referenced through this association, the analysis marks the association as `read-only`. The inference is based on points-to analysis, which determines the set of objects a reference variable or a reference object field may point to. For ownership inference, the points-to analysis solution is needed to approximate the possible accesses between run-time objects; for immutability inference, it is needed to approximate the objects modified within a method. It is important to note that the analyses can work on complete programs (i.e., whole programs) as well as on incomplete programs (i.e., software components). This paper focuses on complete programs in order to (i) clearly present the underlying algorithms, and (ii) emphasize analysis scal-

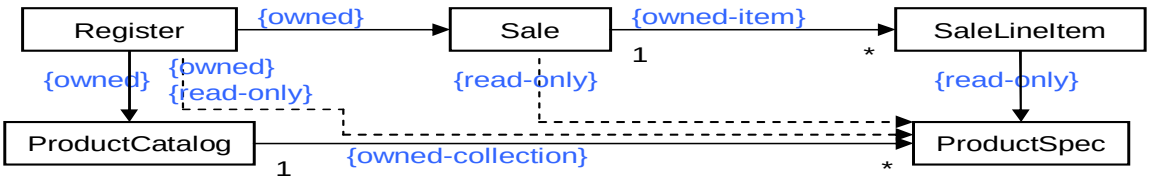

Figure 1: Ownership and immutability constraints on UML associations.

ability on large programs.

Surprisingly, while applying our analysis on the code from a popular textbook [23], we discovered a serious bug in this code. Furthermore, we present empirical results on several relatively large Java benchmarks. In our experiments, on average 28% of the reverse-engineered associations were determined to be `owned`, and 27% were determined to be `read-only`. We present a precision evaluation which indicates that the analyses achieve adequate precision—the ownership inference almost never misses an `owned` association and the immutability inference rarely misses a `read-only` association. In addition, the ownership and immutability analyses are practical, running in less than 7 minutes on all but one benchmark. These results indicate that the analyses are precise and practical and can be incorporated in a software tool for the reverse engineering of UML class diagrams. They can effectively support verification of ownership and immutability which will lead to high quality, secure, understandable and maintainable software systems.

This work has the following contributions:

- We propose a new mechanism for alias control. It is based on the UML and light-weight verification of properties related to software quality and security.
- We develop implementation-level models for ownership and immutability that capture well the meaning of these concepts in design.
- We develop novel static ownership and immutability inference analyses.
- We present an empirical study on relatively large Java programs which demonstrates that the analyses are adequately precise and practical.

The rest of the paper is organized as follows. Section 2 presents an example that motivates the idea of UML-based alias control. Section 3 presents the ownership and immutability models and formulates the analysis problem. Section 4 presents the inference analyses and Section 5 presents the empirical results. Section 6 discusses related work and Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

This section motivates the idea of UML-based alias control. Consider the UML class diagram in Figure 1. It illustrates the design of a supermarket Point-of-Sale system and is taken directly from a popular textbook on software design and the UML [23]. The solid lines represent permanent associations (implemented through instance fields), and the dashed lines represent temporary dependencies (typically implemented through local variables). We have added ownership and immutability constraints based on the description in the textbook—these constraints formalize the design principles emphasized in the textbook.

A singleton `Register` object, an abstraction for the cash register, controls the sale logic. It creates a `ProductCatalog` object that stores the specifications of all products (i.e., the `ProductSpec` objects). The `Register` object creates a `Sale` object, initiates the sale, passes information about sale items to the `Sale` object and completes the sale. When a new sale item is processed, the `Register` fetches the corresponding `ProductSpec` object from the catalog, and passes that object to the `Sale` object. The `Sale` object creates a new `SaleLineItem` object for the current sale item and passes the `ProductSpec` object to it.

The association from `Register` to `Sale` is marked as `owned`. Thus, the `Register` *owns* each `Sale` object it refers through this association—intuitively, the `Register` may create a `Sale` object, pass it to other parts of its representation, but cannot leak the `Sale` object to outside parts (e.g., objects that are part of the User Interface (UI) of the system). The other `owned` constraints have analogous meaning. Furthermore, the association between `SaleLineItem` and `ProductSpec` is marked as `read-only`. Thus, the `SaleLineItem` object cannot modify the `ProductSpec` object it refers to. `Register` and `Sale` objects have `read-only` access to `ProductSpec` objects as well, leaving the `ProductCatalog` the only object that can initialize and update product information. Note that for one-to-many associations (e.g., `ProductCatalog` to `ProductSpec`) one can specify constraints on the collection and on the items. For example, a ProductCatalog owns the collection that stores `ProductSpecs`, but does not own the `ProductSpec` items stored in this collection.

The ownership and immutability constraints inherently support reasoning about software design principles such as "Low Coupling", "Information Expert", etc. [23]. For example, the constraint that `Register` owns the `Sale` objects forbids coupling from UI classes to `Sale` which helps achieve low coupling and separation of the UI layer from the domain layer. The constraint that `SaleLineItem` has read-only access to `ProductSpec` forbids `SaleLineItems` from modifying `ProductSpecs`; in fact, the `ProductCatalog` is the "information expert" and the only object that can initialize and update product information. Most importantly, the ownership and immutability constraints impose requirements on the implementation. These requirements can be continuously verified in a light-weight manner through reverse engineering of the UML class diagram.

The Java code from [23] that corresponds to this diagram

is given in Appendix A. Surprisingly, when we applied our analysis on this code, the association between `SaleLineItem` and `ProductSpec` was reported as non-`read-only`. A brief examination of the code revealed a bug that could be very serious—the `SaleLineItem` object mistakenly modified the *price* field of the `ProductSpec` object. As a result, subsequent sales fetched `ProductSpec`s with wrong prices and computed incorrect sale totals!

In summary, verifying and enforcing ownership and immutability constraints will lead to higher quality, more secure, understandable and maintainable software systems.

3. PROBLEM STATEMENT

Conventionally, software tools reverse engineer UML associations by examining instance fields of reference type in the code (e.g., a reference field f of type B in class A is reverse engineered into an association from A to B labeled with f).[1] In our model, an association through a field f is marked as `owned` if it can be proven that for each run-time edge $o \xrightarrow{f} o'$, o owns o'. An association through f is marked as `read-only` if it can be proven that for each run-time edge $o \xrightarrow{f} o'$, o does not mutate o'. Thus, it remains to give suitable definitions of implementation-level ownership and immutability.

3.1 Ownership Model

The ownership model is based on the notion of *owners-as-dominators* [13, 12, 35]. In this model, each execution point is represented by an *object graph* which shows access relationships between run-time objects. There is an edge $o \rightarrow o'$ from run-time object o to run-time object o' (i.e., we say that o' is accessed in the *context* of o) if and only if one of the following is true:

- Reference instance field f in o refers to o'.
- Object o is an array object with element o'.
- An instance method or constructor invoked on receiver o has local variable r that refers to o', or a static method called from an instance method or constructor invoked on o, has a local variable r that refers to o'.[2]

Note that the first two items account for somewhat permanent, "heap" accesses. In contrast, the last item accounts for temporary, "stack" accesses that appear when a local is set to point to an object, and disappear when the method enclosing the local finishes execution.

In accordance with the owners-as-dominators model, we say that *o owns o′* if and only if o is the immediate dominator of o' in the object graph at all execution points.[3] Consider the object graph in Figure 3; it is a summary graph which includes all object graphs during the execution of `main` in Figure 2. Node `root` represents the start of program execution. The other nodes correspond to the objects created at the appropriate allocation sites in Figure 2. We have that `Vector` does not own `Object[]` because during the execution of `next` there is a stack access from `VIter` to `Object[]`. We would have that `Vector` does own `Object[]` if `next` is never executed (i.e., line 15 is removed from `main`).

The ownership inference problem therefore is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$ o owns o' according to the above definition. The associations through these fields are marked as `owned`.

```
public class Vector {
 protected Object[] data;
 public Vector(int size) {
1 data = new Object[size]; }
 public void add(Object e,int at) {
2 data[at] = e; }
 public Object elementAt(int at) {
3 return data[at]; }
 public Iterator iterator() {
4 return new VIterator(this); }
}
final class VIterator implements Iterator {
 Vector vector;
 int count;
 VIterator(Vector v) {
5  this.vector = v;
6  this.count = 0; }
 Object next() {
7  Object[] data = vector.data;
8  int i = this.count;
9  this.count++;
10 return data[i]; }
}
main() {
11 Vector v = new Vector(100);
12 X x = new X();
13 v.add(x,0);
14 Iterator i = v.iterator();
15 x = (X) i.next();
16 x.m();
}
```

Figure 2: Simplified vector and its iterator.

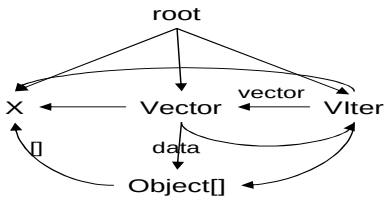

Figure 3: Object graph for Figure 2.

3.2 Immutability Model

Let e be an execution of a method m on receiver object o; e modifies an object o' if and only if it triggers a change in the object structure rooted at o'—that is, e leads to a statement $p.f = q$ which modifies an object o'' reachable from o'. For example, the execution of method `add` with receiver `Vector` (line 13 in Figure 2) modifies `Object[]`. We say that *o has read-only access to o′* if no execution of a method m on receiver o modifies o'. Thus, in the above example `Vector`

[1]The rest of the paper focuses on permanent associations (implemented with instance fields). Although our models and analyses are general and can handle temporary dependencies, we omit their discussion for clarity.

[2]We require that there is explicit reference variable for every accessed object (i.e., statements $r.m().n()$ are re-written into an equivalent sequence $r_1 = r.m()$; $r_1.n()$;).

[3]Node m *dominates* node n if every path from the root of the graph that reaches node n has to pass through node m. The root dominates all nodes. Node m *immediately dominates* node n if m dominates n and there is no node p such that m dominates p and p dominates n.

does not have read-only access to `Object[]`.

The model does not treat constructor invocations and the corresponding initialization statements `this.f=q` as modifications of the newly constructed object. This is done in order to capture better the intuitive meaning of immutability in the context of class diagrams.

The immutability inference problem therefore is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$ o has read-only access to o'. The associations through these fields are marked as `read-only`.

4. OWNERSHIP AND IMMUTABILITY ANALYSES

The ownership and immutability analyses can be applied on complete programs, as well as on incomplete programs (i.e., components); intuitively, the whole-program analysis can be adapted to work on incomplete programs by utilizing a technique called *fragment analysis* [37, 40, 38, 27]. We present the analyses in the whole-program setting in order to (i) emphasize the underlying algorithms, and (ii) demonstrate scalability on real, large-size Java benchmarks.

The ownership and immutability analyses are built as independent clients of a *points-to analysis*. Points-to analysis determines the set of objects that a given reference variable or a reference object field may refer to. For the purposes of the ownership client, points-to analysis information is needed to construct a graph that approximates all possible object graphs that can happen during program execution; subsequently the graph is used to reason about ownership. For the purposes of the immutability client, points-to information is used to approximate the objects mutated in the methods of a class, and subsequently reason about immutability of fields. There is a large body of work on points-to and related class analysis with different trade-offs between cost and precision. [31, 33, 4, 6, 17, 41, 46, 48, 47, 26, 39, 16, 28, 29, 51, 24, 8, 52]. For this work we consider ownership and immutability inference based on the well-known Andersen-style flow- and context-insensitive points-to analysis for Java [39, 51, 24].[4]

4.1 Points-to Analysis

The points-to analysis is defined in terms of three sets. Set R is the set of locals, formals and static fields of reference type. Set O is the set of object names; the objects created at an allocation site s_i are represented by object name $h_i \in O$. Set F contains all instance fields in program classes. The analysis solution is a *points-to graph* where the edges represent the following "may-refer-to" relationships.

- Let $r \in R$ and $h \in O$. An edge (r, h) in the points-to graph means that at run time r may refer to some object that is represented by h.
- Let $f \in F$ be a reference instance field in objects represented by some $h \in O$. An edge $(h.f, h_2)$ means that at run time field f of some object represented by h may refer to some object represented by h_2.
- Let h represent array objects. An edge $(h[], h_2)$ shows that some element of some array represented by h may refer at run time to an object represented by h_2.

The Andersen-style points-to analysis is an inclusion-based analysis. It propagates may-refer-to relationships by analyzing program statements. For example, when it analyzes statement "$p = q$" it infers that p may refer to any object that q may refer to.

For the rest of the paper we use notation o to refer to run-time objects (e.g., o, o', o_i, etc.); we use notation h to refer to analysis names that abstract the run-time objects (e.g., h, h', h_i, etc.).

4.2 Ownership Client

The output of the points-to analysis is needed to construct the *approximate object graph* Ag which approximates all possible run-time object graphs. Subsequently, Ag is used for ownership inference.

4.2.1 Approximate Object Graph

The nodes in Ag are taken from the set of object names O and the edges represent "may-access" relationships. Figure 4 outlines the construction of Ag given a points-to graph Pt. Intuitively, the algorithm tracks flow of objects from one context to another context. Lines 1-2 account for edges due to object creation; at object allocation sites (i.e., constructor calls), the newly created object becomes available in the context of the caller. The contexts of the caller m are stored in set $\mathcal{C}_m$. If m is an instance method, $\mathcal{C}_m$ equals to the points-to set of the implicit parameter `this` of m. If m is a static method, $\mathcal{C}_m$ includes the points-to sets of all implicit parameters `this` of instance methods n reachable backwards from m on a chain of static calls (i.e., $\mathcal{C}_m$ includes the closest instance contexts enclosing m); if `main` is reachable backwards from m on a chain of static calls, $\mathcal{C}_m$ includes the special context `root` as well. At allocation sites new edges are added to Ag from each context of the enclosing method m to the object name that represents the newly created object. Lines 3-4 account for edges due to flow out from other contexts to the context of m. For example, at an instance call not through `this` new edges are added from each context of m to each returned object. Finally, lines 5-6 account for edges due to flow from the contexts of m into other contexts. For example, at an instance call, edges are added from each object in the points-to set of the receiver to each object in the points-to set of a reference argument. Finally, line 7 labels with field identifier f each edge $h_i \rightarrow h_j \in Ag$ for which there is an edge $(h_i.f, h_j) \in Pt$.

As an example, consider the code in Figure 2. In this case, the algorithm in Figure 4 constructs precisely the summary run-time object graph in Figure 3. Edges `root`→`Vector`, `root`→`X` and `Vector`→`Object[]` are due to code lines 11, 12 and 1 respectively (lines 1-2 in the algorithm). Edge `Vector`→`X` is due to code line 13 (lines 5-6 in the algorithm). Edge `Object[]`→`X` is due to code line 2. Furthermore, edge `root`→`VIter` is due to code line 14, and edges `Vector`→`VIter` and `VIter`→`Vector` are due to line 4. Finally, edges `VIter`→`Object[]` and `VIter`→`X` are due respectively to lines 7 and 10 in `next`.

[4]Flow-insensitive analyses do not take into account the flow of control between program points and are less precise and less expensive than flow-sensitive analyses. Context-sensitive analyses distinguish between different calling contexts of a method and are more precise and more expensive than context-insensitive ones.

input *Stmt*: set of statements Pt: $R \cup O \rightarrow \mathcal{P}(O)$
output $Ag : O \rightarrow \mathcal{P}(O)$
[1] **foreach** statement s in method m
$s_i : l = new\ C(...)$
[2] add $\{c \rightarrow h_i \mid c \in \mathcal{C}_m\}$ to Ag
//**creation** flow into caller contexts
[3] **foreach** statement s in method m
$s : l = r.n(...)$ s.t. $r \neq$ `this`,
$s : l = r.f$ s.t. $r \neq$ `this`
[4] add $\{c \rightarrow h_j \mid c \in \mathcal{C}_m \wedge (l, h_j) \in Pt\}$ to Ag
//**outflow** from callee contexts to caller contexts
[5] **foreach** statement s in method m
$s : l = new\ C(r)$,
$s : l.n(r)$ s.t. $l \neq$ `this`,
$s : l.f = r$ s.t. $l \neq$ `this`
[6] add $\{h_i \rightarrow h_j \mid (l, h_i) \in Pt \wedge (r, h_j) \in Pt\}$ to Ag
//**inflow** into callee contexts from caller contexts
[7] label with f each $h_i \rightarrow h_j \in Ag$ s.t. $(h_i.f, h_j) \in Pt$

Figure 4: Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty.

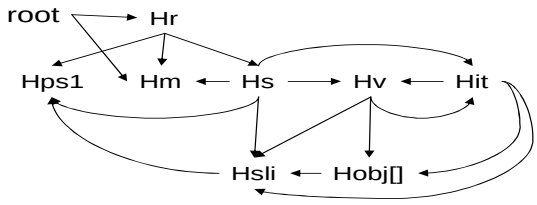

Figure 5: Partial Ag for Section 2.

The object graph construction and ownership inference need to consider two special cases: (i) static fields and (ii) self-references (i.e., when an object references itself through `this` as in `r.m(this)`). For brevity, we do not discuss these cases; our implementation handles them correctly.

4.2.2 Ownership Inference

The ownership inference uses Ag to reason about object ownership. Consider the object graph in Figure 5, extracted from the code for Section 2 from [23]; the code is given in Appendix A. Node `root` represents the special context of `main` and node *Hr* represents the `Register` object (created in `main`). *Hps1* represents `ProductSpec` objects (created in `ProductCatalog`), *Hm* represents `Money` objects (created in `main`), and *Hs* represents `Sale` objects (created in `Register.makeNewSale`). *Hsli* represents `SaleLineItem` objects (created in `Sale.makeLineItem`) and *Hv* represents the collection needed to store the `SaleLineItems`. Finally, *Hit* represents iterators over the collection of `SaleLineItems` (used in `Sale.getTotal`).

The inference analysis (Figure 6) examines an edge $h_i \rightarrow h_j$ in the object graph and attempts to prove that for *each* run-time instance $o_i \rightarrow o_j$ of that edge o_i dominates o_j; intuitively, it reasons about the flow of run-time objects based on the object graph abstraction of this flow. The inference is based on the following intuition: an object o_j can flow from o_i into some o_k only if one of the following is true: (1) o_k has a handle to both o_i and o_j (and hence Ag contains edges $h_k \rightarrow h_i$, $h_k \rightarrow h_j$ and $h_i \rightarrow h_j$), or (2) o_i has a handle to both h_k and h_j (and hence Ag contains edges $h_i \rightarrow h_k$, $h_i \rightarrow h_j$, $h_k \rightarrow h_j$). Thus, to track flow, the analysis considers edge triples $h_m \rightarrow h_o$, $h_m \rightarrow h_n$, and $h_n \rightarrow h_o$ in Ag. In Figure 5 edge triple *Hr→Hps1*, *Hr→Hs*, *Hs→Hps1* represents the fact that a `ProductSpec` object flows into a `Sale` object from the `Register` object. Note that if an edge $h_i \rightarrow h_j$ in Ag does not have an h_k such that either (1) h_k has handles to both h_i and h_j, or (2) h_i has handles to both h_k and h_j, we have that each o_i exclusively owns each o_j it refers to (i.e., o_i is the only object that has a reference to o_j). In Figure 5 *Hr→Hs* is such an edge; it represents that the `Register` exclusively owns the `Sale` objects it creates.

input Ag: $O \rightarrow \mathcal{P}(O)$ $h_i \rightarrow h_j$: $O \times O$
output *Closure*: $O \rightarrow \mathcal{P}(O)$, isClosed: boolean
[0] if $isOutside(h_i \rightarrow h_j)$ **return** false
[1] $Closure = \{h_i, h_j\}$, $W = \{h_i\}$
[2] while W not empty
[3] take h_k from W
[4] foreach $h_m \in Tgts(h_k) \cap Closure$
[5] foreach $h_n \in Tgts(h_k) \cap Srcs(h_m)$, $h_n \notin Closure$
[6] if $isOutside(h_i \rightarrow h_n)$ **return** false
[7] if $valid(h_k, h_n, h_m)$ add h_n to *Closure* and to W
[8] foreach $h_m \in Srcs(h_k) \cap Closure$
[9] foreach $h_n \in Tgts(h_m) \cap Srcs(h_k)$, $h_n \notin Closure$
[10] if $isOutside(h_i \rightarrow h_n)$ **return** false
[11] if $valid(h_m, h_n, h_k)$ add h_n to *Closure* and to W
[12] **return** true

procedure *valid*
input h_i, h_k, h_j, where $h_i \rightarrow h_j$, $h_i \rightarrow h_k$, $h_k \rightarrow h_j$
output isValid: boolean
[1] if $isIn(h_k \rightarrow h_j)$ and $h_i \in In(h_k \rightarrow h_j)$ **return** true
[2] if $isOut(h_i \rightarrow h_j)$ and $h_k \in Out(h_i \rightarrow h_j)$ **return** true
[3] **return** false

Figure 6: Ownership inference: computing the closure of edge $h_i \rightarrow h_j$. $Tgts(h)$ stands for $\{h' \mid h \rightarrow h' \in Ag\}$ and $Srcs(h)$ stands for $\{h' \mid h' \rightarrow h \in Ag\}$.

Consider the algorithm in Figure 6, lines 0 to 12, assuming that *valid* always returns true; the role of *valid* will be explained in Section 4.2.3. The algorithm makes use of a predicate $isOutside(h_i \rightarrow h_j)$ (lines 0, 6 and 10)—an edge $h_i \rightarrow h_j$ is an *outside edge* if there exists an h_k such that h_k has handles to both h_i and h_j. Intuitively, *isOutside* conservatively captures the situation when some o_j flows from (or into) an "outside" object o_k and therefore there might be an access path to o_j that does not pass through o_i. In Figure 5, edge *Hs→Hm* is an outside edge. It capture the situation that a `Money` object is passed from the `Register` to a `Sale`; clearly, the `Sale` object does not own the `Money` object. If the edge that is examined, namely $h_i \rightarrow h_j$, is not an outside edge, the algorithm proceeds to compute the *Closure* of $h_i \rightarrow h_j$. The algorithm examines the paths from h_i to h_j. If at some point it detects a path that originates in an outside edge, it returns false (lines 6 and 10) and the partially computed closure. Otherwise, it returns true and computes the entire closure—the closure represents *all* paths from o_i to o_j for *each* run-time edge $o_i \rightarrow o_j$; it is guaranteed that all paths from o_i to o_j are internal, and thus o_i owns o_j.

We illustrate the algorithm with edge *Hr→Hps1* in Figure 5. At the first iteration of the while loop h_k is *Hr* and *Hs* is added to *Closure* and the worklist due to lines 4-7. At the second iteration, h_k is *Hs* and *Hsli* is added due to lines 4-7. At the third iteration, h_k is *Hsli*. In this case, lines 4-7 do not yield new nodes; lines 8-11 however, yield two new nodes, *Hv*, and *Hit*. At the next iteration h_k is *Hv* and lines 4-7 yield node *Hobj[]*. At the next two iterations no new nodes are added and the algorithm returns true. The closure consists of nodes *Hr, Hs, Hsli, Hv, Hit, Hobj[]* and *Hps1*, and the edges between them.

Let the algorithm return true for some edge $h_i \rightarrow h_j$. Then we have that for each $o_i \rightarrow o_j$ represented by $h_i \rightarrow h_j$, o_i dominates o_j.

We give the correctness argument for this statement. Let $o_i \rightarrow o_j$ be an edge represented by $h_i \rightarrow h_j$ and let $\mathcal{K}(o_i \rightarrow o_j)$ denote the paths from o_i to o_j at execution point p, whose representative is in $Closure(h_i \rightarrow h_j)$. Suppose that for some $o_k \in \mathcal{K}$ there is a path `root`... $\rightarrow o_x \rightarrow o_k$ that does not pass through o_i—that is, we have that o_i does not dominate o_k. We say that o_k is an *exposed node*. Without loss of generality we may assume that o_x is not in $\mathcal{K}$ (clearly, if we continue on the path we will reach `root` which is not in $\mathcal{K}$). The only way o_k may flow from $\mathcal{K}$ into o_x (or from o_x into $\mathcal{K}$) is through a "parent"—that is, there must be an edge triple such that $o_n \in \mathcal{K}$, and $o_x \rightarrow o_n$, $o_x \rightarrow o_k$ and $o_n \rightarrow o_k$. To see this, suppose that o_k flows to (or from) o_x in the other possible way, through a "sibling", and there we have $o_n \in K$ and $o_n \rightarrow o_x$, $o_n \rightarrow o_k$ and $o_x \rightarrow o_k$. This is impossible, because the representatives of these edges would have been processed at either lines 4-7 or 8-11, and thus we would have the representative of o_x in *Closure*, and thus o_x in $\mathcal{K}$. We say that edge $o_n \rightarrow o_k \in \mathcal{K}$ is an *exposed edge*.

Thus, o_n is an exposed node as well. Making the argument that node o_n can only be exposed through a "parent" leads to an exposed edge $o_{n-1} \rightarrow o_n$. Repeating the argument leads to an exposed edge $o_i \rightarrow o_1$ whose representative $h_i \rightarrow h_1$ is an outside edge. Thus, there is a contradiction because if the computation of *Closure* encounters such an edge, the algorithm returns false (lines 6 and 10).

Next, note that there is an order of exposure—that is, when o_k is exposed through node o_n we must have that o_n is exposed *before* o_k. Clearly, if there is a cycle of exposed edges, the cycle is ordered as well (i.e., edge $o_{n-1} \rightarrow o_n$ was exposed *before* $o_n \rightarrow o_k$, etc.). Therefore, some node must be exposed due to an exposed edge *not* in the cycle. Since at execution point p there are finite number of nodes and edges in $\mathcal{K}$, the exposed path will reach o_i.

If the algorithm in Figure 6 returns true for every edge labeled with f, the ownership analysis concludes that the association through f is `owned`.

4.2.3 Improved Ownership Inference

Note that the analysis, as described above will likely incur substantial imprecision and cost. This is due to the fact that not all edge triples $h_i \rightarrow h_j$, $h_i \rightarrow h_k$, $h_k \rightarrow h_j$ represent valid flow. For example, suppose that edges $h_i \rightarrow h_j$ and $h_k \rightarrow h_j$ are due to object creation (lines 1-2 in the algorithm in Figure 4) and $h_i \rightarrow h_k$ is due to inflow (lines 5-6). Clearly, edges $h_i \rightarrow h_j$ and $h_k \rightarrow h_j$ refer to two distinct run-time objects that are represented with the same name, h_j. However, the analysis concludes that there might be an

input Pt: $R \rightarrow \mathcal{P}(O)$
output Mod: $m \rightarrow \mathcal{P}(R)$
[0] foreach instance field write s: $p.f = q$
  where `p`$\neq$`this` OR $EnclMethod(s)$ is not a constructor
[1] add p to $Mod(EnclMethod(s))$
[2] while changes occur in Mod
[3] foreach call s: $C.m()$ or $r.m()$
[4] foreach target m' of the call
[5] add $Mod(m')$ to $Mod(EnclMethod(s))$

input $h_i \rightarrow h_j \in O \times O$ $Mod : m \rightarrow \mathcal{P}(R)$
output readOnly: boolean
[6] foreach call s: $r.m(...)$ s.t. $r \neq$ `this` and $h_i \in Pt(r)$
[7] if $TrClosure(h_j) \cap Pt(Mod(target(h_i, m))) \neq \emptyset$
[8] return false
[9] return true;

Figure 7: Immutability inference: computing the read-only status of $h_i \rightarrow h_j$.

o_j that flows from some o_i into some o_k and erroneously infers that edge $h_i \rightarrow h_j$ is not owned. Invalid triples affect not only precision but cost as well. In the above example, when reasoning about edge $h_i \rightarrow h_j$ the analysis needs to reason about edges $h_i \rightarrow h_k$ and $h_k \rightarrow h_j$, which is clearly redundant as these edges are irrelevant to $h_i \rightarrow h_j$.

The edges in the object graph may be characterized as *creation* (due to lines 1-2 in Figure 4), *outflow* (due to lines 3-4) and *inflow* (due to lines 5-6). First, let $h_i \rightarrow h_j$ be an outflow edge. A triple with h_k (i.e., $h_i \rightarrow h_j$, $h_i \rightarrow h_k$ and $h_k \rightarrow h_j$) will be a valid triple only if for some statement $l = r.n()$ that produces outflow edge $h_i \rightarrow h_j$ we have that r point to h_k. Second, let $h_k \rightarrow h_j$ be an inflow edge. A triple with h_i (i.e., $h_k \rightarrow h_j$, $h_i \rightarrow h_k$ and $h_i \rightarrow h_j$) will be a valid triple only if for some statement $l.m(r)$ that produces this edge we have that the `this` pointer of the enclosing method of $l.n(r)$, point to h_i.

The algorithm in Figure 4 is augmented to track valid sources for outflow and inflow edges. Lines 4' and 6' below are added respectively after lines 4 and 6; there is a set *Out* for each outflow edge and a set *In* for each inflow edge.

[4'] add $Pt(r)$ to $Out(c \rightarrow h_j)$
[6'] add $Pt(\texttt{this}_m)$ to $In(h_i \rightarrow h_j)$

Subsequently, the ownership inference in Figure 6 uses procedure *valid* to filter out invalid triples. For example, if $h_k \rightarrow h_j$ is an inflow edge, h_i must appear in $In(h_k \rightarrow h_j)$. The actual implementation stores variables (i.e., r and `this`) rather than objects in the *In* and *Out* sets which turns out to be more efficient in terms of memory; procedure valid checks the points-to sets of the stored variables.

4.3 Immutability Client

4.3.1 Immutability Inference

The immutability inference is presented in Figure 7. Lines 0-5 perform standard side-effect analysis [43, 28, 38] which computes a *Mod* set for each method m. Lines 0-1 process each statement s: $p.f = q$ and store p in the *Mod* set for the enclosing method of s. Subsequently, lines 2-5 propagate the *Mod* sets backwards on the call graph. Set $Mod(m)$ contains all reference variables p on the left-hand-side of an instance field write, reachable on a call chain from m. The union of

```
class A {
 B b1;
 A(B _b1) { b1 = _b1; ...  }
 m() { B b2 = new B(); b2.setField(10); ...  }
 }
main() {
 B b1 = new B(); b1.setField(5);
 A a = new A(b1); a.m();
 }
```

Figure 8: Imprecision of immutability inference.

the points-to sets of these variables approximates the set of objects that may by modified during the invocation of m.

Finally, lines 6-9 take an edge $h_i \rightarrow h_j \in Ag$ as input and attempt to show that for all run-time edges $o_i \rightarrow o_j$ represented by this edge o_i has read-only access to o_j. The analysis examines each method call $r.m(...)$ on receiver h_i (i.e., $h_i \in Pt(r)$). $TrClosure(h_j)$ denotes the transitive closure of h_j on the points-to graph—that is, the set of all nodes reachable from h_j on a path of field edges. $Pt(S)$ extends the Pt notation over sets as follows: $Pt(S) = \bigcup_{p \in S} Pt(p)$. If for some call the transitive closure of h_j intersects with the set of modified objects of the run-time target of the call (i.e., $target(h_i, m)$), the analysis determines that edge $h_i \rightarrow h_j$ is not immutable. If this intersection is always empty, the analysis determines that $h_i \rightarrow h_j$ is immutable.

Consider the Point-of-Sale code in Appendix A. In method `getSubtotal` in class `SaleLineItem`, `spec` points to *Hps1* and field *price* of *Hps1* points to *Hm1* (*Hm1* represents the `Money` object that holds the price of the product). Thus, `getSubtotal` calls method `times` on *Hm1*. The analysis correctly determines that *Mod*(`times`) equals {`times.this`}—that is, `times` *changes* the value of the receiver object. Thus, *Mod*(`getSubtotal`) equals {`times.this`} and we have that *Hm1* is included in set $Pt(Mod(\texttt{getSubtotal}))$.

Now consider the call to `getSubtotal` in `getTotal` in class `Sale`, and the effect of this call on edge $Hsli \overset{spec}{\rightarrow} Hps1$. The intersection of the set of objects modified by `getSubtotal` and the transitive closure of *Hps1* is non-empty; it includes *Hm1*. The analysis determines that a `SaleLineItem` object can modify a `ProductSpec` object which is a violation of the immutability constraint specified in Figure 1. Further examination revealed that this was a serious bug in the code in [23]; it caused subsequent sales to fetch wrong product prices and compute incorrect totals. The bug can be fixed by changing method `getSubtotal` as follows:

```
Money subtotal = new Money(spec.getPrice());
return subtotal.times(quantity);
```

If the procedure for checking an edge returns true for every edge labeled with f, the immutability analysis concludes that the association through f is `read-only`.

4.3.2 Improved Immutability Inference

It is easy to see that the algorithm in Figure 7 may incur substantial imprecision. Consider the code in Figure 8. Field $b1$ is immutable in A. The B object created in `main` and referred by field $b1$ is denoted by name H_{b1}, and the B object created in `m` is denoted by H_{b2}. *Mod*(`setField`) equals {`setField.this`}; it is propagated to *Mod*(`m`) and we have that *Mod*(`m`) equals {`setField.this`} as well. The points-to set of `setField.this` contains both H_{b1} and H_{b2} and the analysis concludes imprecisely that $b1$ is mutable.

To improve the analysis we introduce a limited form of context sensitivity. The main idea is that when propagating the *Mod* set of the callee (line 5), the analysis "maps" modified formal parameters to their corresponding actuals. More precisely, it examines every variable $v \in Mod(m')$. If v is an unassigned formal parameter of m', v is mapped to the corresponding actual at the call and the actual is added to $Mod(EnclMethod(s))$; otherwise v itself is added to $Mod(EnclMethod(s))$.[5] Consider again the code in Figure 8. When propagating *Mod*(`setField`) to *Mod*(`m`) the analysis maps `setField.this` to the actual argument at the call, namely variable `b2`. As a result *Mod*(`m`) equals {`b2`}. Since `b2` points to H_{b2} only, the intersection of the transitive closure of H_{b1} and $\{H_{b2}\}$ is empty and the analysis concludes that $b1$ is immutable in A.

It is easy to see that this improvement preserves the correctness of the analysis. In the same time, we observed that it had substantial impact on precision. This is due to the fact that objects are typically modified through setters and statements `this`.f=... and the inexpensive technique presented in this section targets precisely such mutations.

4.4 Analysis Complexity

Let N be the size of the program being analyzed—that is, the number of statements, the number of object names and the number of variables is of order N. The complexity of the underlying Andersen-style points-to analysis is $O(N^3)$. It remains to analyze the ownership and immutability clients.

The complexity of the ownership client is dominated by the ownership inference in Figure 6. For each edge $h_i \rightarrow h_j$ there are at most $O(N)$ objects h_k that are processed on the worklist. Furthermore, for each o_k there are at most $O(N)$ h_j objects and at most $O(N)$ h_n objects (processed at lines 4-7 and at lines 8-11). Thus, for each edge the analysis does $O(N^3)$ work. There are $O(N^2)$ edges and therefore the complexity of ownership inference is $O(N^5)$.

The complexity of the immutability client is dominated by the checking of edges (lines 6-9 in Figure 7). The computation of the transitive closures of all nodes is $O(N^3)$. For each edge $h_i \rightarrow h_j$ the analysis processes at most $O(N)$ calls. For each call it does $O(N)$ work at line 8 checking whether each $h' \in Pt(Mod(target(h_i, m)))$ is in $TrClosure(h_j)$. Since there are $O(N^2)$ edges, the complexity of immutability inference is $O(N^4)$. The entire analysis is dominated by the ownership client and therefore has complexity $O(N^5)$.

5. EXPERIMENTAL RESULTS

The goal of the empirical study is to address three questions. First, do the analyses scale to large Java applications? Second, how often do our analyses discover owned and immutable fields? Third, how *imprecise* the analyses are—that is, how often they miss owned or immutable fields?

The ownership and immutability clients are implemented in Java using the Soot 2.2.3 [50] and Spark [24] frameworks; specifically they are implemented as clients of the Andersen-style points-to analysis provided by Spark. We performed whole-program analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a 900MHz Sun Fire 380R machine with 4GB of RAM. The implementation which includes Soot and Spark was run with a max heap size of 1GB.

[5]Implicit parameter `this` cannot be assigned, and other formal parameters are rarely assigned.

(1)Program	(2)Description	(3)Size		
		#User Classes	#User Methods	#Reachable Methods
`jdepend-2.9.1`	A quality metrics suite for Java	17	225	3962
`javad`	Classfile decompiler	41	156	3838
`JATLite-0.4`	Template for writing software agents	45	442	6279
`undo`	Undo functionality for sysadmins	237	1709	5644
`hsqldb-1.8.0`	Relational database engine and tools	196	3743	7177
`soot-c`	Analysis framework for Java	579	2935	6046
`sablecc-j`	Java parser generator	300	2024	7970
`polyglot-1.3.2`	Framework for Java language extensions	267	3418	7449
`antlr`	Parser and lexical analyzer generator	126	1738	5102
`bloat`	Java bytecode optimizer	289	3232	6402
`jython`	Python interpreter	163	2892	5606
`pmd`	Java source code analyzer	718	7057	8653
`ps`	Postscript interpreter	200	908	5396

Table 1: Information about the Java benchmarks.

Native methods are handled by utilizing the models provided by Soot. Reflection is handled by specifying the dynamically loaded classes which Spark uses to appropriately resolve reflection calls. This approach is used in other whole-program analyses based on Soot and Spark [25, 45].

Our benchmark suite includes several relatively small applications, `soot-c` and `sablecc-j` from the Ashes suite [1], several benchmarks from the DaCapo benchmark suite version beta051009 [2] and the Polyglot Java front-end. The suite is described in Table 1. The number of user classes and user methods fetched by Soot are shown in the first two columns of multicolumn (3); these numbers exclude the standard libraries but include other libraries shipped with the application. The last column shows the number of methods (user and library), determined to be reachable by Spark.

5.1 Results

We applied the ownership and immutability inference algorithms on instance fields of reference type in user classes to determine which fields accounted for `owned` associations and which accounted for `read-only` associations.[6] Table 2 shows the running time of the analysis. The first column shows the running time for Soot and Spark, and the two subsequent columns show the running times for the ownership and immutability clients. Clearly, our analyses scale well, even on applications with close to 9000 reachable methods. The combined time for ownership and immutability analysis does not exceed 7 minutes on twelve out of thirteen benchmarks; on the most expensive benchmark, `pmd`, it still runs in under 11 minutes.

The first column of Table 3 shows the number of reference instance fields in user classes. On average, the ownership analysis identified 28% of the fields as owned (column #Owned).Also, on average, the immutability analysis identified 27% of the fields as read-only (column #Immutable).

5.2 Analysis Precision

The issue of analysis precision is of crucial importance for software tools. For example, if the ownership analysis is imprecise, it may report that an association is non-owned while in reality it is owned (i.e., the analysis reports that certain representation may be exposed while in fact it is not). Similarly, the immutability analysis may report that an association is non-read-only, while in reality it is. Such information is not useful and may confuse the user. For example, if a user attempts to verify lack of representation exposure, imprecision will mean that potentially large amount of code will have to be examined manually. Therefore, imprecision must be carefully evaluated by analysis designers.

Program	Points-to Analysis	Ownership Analysis	Immutability Analysis
`jdepend`	1m35s	32s	10s
`javad`	1m33s	27s	3s
`JATLite`	2m37s	1m29s	35s
`undo`	3m3s	1m52s	37s
`hsqldb`	2m57s	2m15s	2m31s
`soot`	2m23s	1m13s	1m38s
`sablecc`	3m5s	1m49s	1m30s
`polyglot`	9m39s	2m44s	3m38s
`antlr`	2m25s	1m4s	35s
`bloat`	2m36s	1m57s	3m8s
`jython`	1m58s	1m21s	3m9s
`pmd`	4m17s	2m22s	8m16s
`ps`	2m19s	1m51s	29s

Table 2: Analysis times.

We performed a study of absolute precision [40, 38, 27] on a subset of the fields. Specifically, we considered all fields in the two smallest benchmarks, `jdepend` and `javad`, and all fields in the class with the largest number of fields for the four largest benchmarks, `hsqldb, polyglot, sablecc` and `pmd` (the size metric that we used was the number of reachable methods, shown in Column (3) of Figure 1). This accounted for a set of 153 instance fields. For this set, we examined manually *each* non-owned field and attempted to prove exposure (i.e., that there is an execution such that an object stored in this field would be exposed outside of its enclosing object). In *all cases* we were able to show exposure—that is, for this set of fields the ownership analysis

[6]Our experiments exclude fields of type `String` because they do not correspond to associations in the UML class diagram.

Program	#Fields	#Owned	#Immutable
jdepend	33	19 (58%)	6 (18%)
javad	40	19 (48%)	40 (100%)
JATLite	142	35 (27%)	13 (9%)
undo	325	73 (22%)	162 (50%)
hsqldb	383	89 (23%)	70 (18%)
soot	340	77 (23%)	57 (17%)
sablecc	304	30 (10%)	40 (13%)
polyglot	435	51 (12%)	92 (21%)
antlr	161	45 (28%)	25 (16%)
bloat	529	81 (15%)	73 (14%)
jython	215	69 (32%)	21 (10%)
pmd	914	318 (35%)	162 (18%)
ps	19	7 (37%)	8 (42%)
Average		28%	27%

Table 3: Ownership and immutability results.

achieved perfect precision. Similarly, we examined each non-read-only field and attempted to prove mutability (i.e., that there is an execution for which an object stored in this field will be mutated by its enclosing object). In *all but 7 cases* we were able to show mutability—that is, the immutability analysis achieved very good precision as well.

In short, the majority of imprecisions are due to context-insensitive object naming in the underlying points-to analysis. However, using context-sensitive object naming, which is expensive, may not be justified—it will likely improve precision only marginally over the current practical analysis which is already adequately precise.

The imprecisions in the immutability analysis were due to two reasons: (i) context-insensitive object naming in the underlying points-to analysis, and (ii) imprecise treatment of polymorphic calls by the immutability inference. We illustrate only the first case, which is more problematic and accounted for 5 of the 7 cases of imprecision; the second case can be addressed by a minor modification of the analysis described in Section 4.3. Recall Figure 8 and consider the following definition of class B:

```
class B {
  C c;
  B() { this.c = new C(); }
  setField(int x) { ... c.setField(x+10); ... }
  }
```

Clearly, field $b1$ is immutable in A. However, the analysis does not distinguish between the C object allocated by H_{b1} and the C object allocated by H_{b2} and it appears that `m` can modify the C object of H_{b1}. This imprecision can be addressed by employing context-sensitive object naming in certain cases. However, using expensive context-sensitive object naming may not be justified—it will likely improve precision only marginally over the current practical analysis which is already adequately precise.

5.3 Conclusions

Our results demonstrate that owned and immutable fields occur often in Java code. The precision study indicates that (1) our ownership and immutability models capture well the notions of ownership and immutability in modeling, and (2) that the analysis is relatively precise and can provide useful information for reverse engineering tools. If integrated in a tool for the reverse engineering of UML class diagrams, the analysis will help verify important security-related properties and help improve software quality and software security.

Overall, the analysis scales well to large programs analyzing close to ten thousand methods in only several minutes. It is significant that the cost of obtaining precise results is practical; clearly, practicality is as crucial as precision for the successful integration of an analysis in a software tool. Note that the analysis works on complete as well as incomplete programs (i.e., software components). Ideally, it will be applied primarily on partially developed subsystems within an iterative development process, and on software components—that is, systems likely substantially smaller than the programs in this study. This study focuses on safe analysis of relatively large complete programs in order to emphasize the scalability and precision of the analysis.

One disadvantage of our current analysis is the need to analyze large libraries. Libraries are typically irrelevant to the properties of interest but in certain cases require significant CPU and memory resources. For example, the immutability client takes 8 minutes on the `pmd` benchmark due to the processing of the transitive closures of library fields that are irrelevant to the immutability of user fields. This problem is inherent in all safe whole-program analyses; it can be solved (or alleviated) by summarizing the effects of library methods, or proving them irrelevant with respect to the properties of interest. This paper focuses on safe whole-program analysis and addressing this problem is beyond its scope.

6. RELATED WORK

The ownership and immutability inference analyses improve substantially upon our previous work on composition inference [27] and side-effect analysis [28] respectively. The main new analysis idea is to employ an inexpensive context-insensitive points-to analysis and improve precision by limited context sensitivity in the clients. This was crucial for precision and scalability; in fact, the old analysis was not only potentially imprecise, but it did not scale beyond the smallest benchmarks in our suite. Further, the analyses are employed towards a new practical purpose—improving the capabilities of UML tools, which will enhance alias control and thus software security and software quality in practice.

Ownership and immutability type systems. Our work is related to work on ownership type systems [30, 13, 5, 12, 10, 22] and work on immutability type systems [20, 32, 9, 49]. Similarly to our work, these articles emphasize the importance of the concepts of ownership and immutability in software development. Unlike our work they focus on type-theoretic approaches and require type annotations provided by the programmer; generally, these approaches require extensions of the language, compiler and run-time environment and therefore will be difficult to adopt in practice. In contrast, our approach uses automatic inference and works directly on Java code; it is based on the universally-known UML and therefore may help advance alias control through ownership and immutability in practice.

Ownership inference. Grothoff et al. [15] present an analysis for Java that infers whether a class is confined within its package. Clarke et al. [14] present a confinement

checking tool for Java, related to [15], that warns against certain kinds of violating program statements; essentially, they solve a checking problem while our analysis solves an inference problem. These analyses work on the class level while our analyses work on the object level. They are more restrictive than ours (e.g., they do not handle pseudo-generic containers well), and do not address the kind of ownership needed for UML-based alias control.

Heine and Lam [19] present an ownership inference algorithm for the purposes of memory leak detection. Their notion of ownership is substantially different than the notion of owners-as-dominators used in our work.

Aldrich et al. [5] present a type inference analysis in accordance with a type system that they develop. Again, our analysis solves a different problem—ownership inference in accordance with the owners-as-dominators model which is different than the type system in [5] (e.g., the `owned` type in [5] captures exclusive ownership only, although access can be allowed through user-specified alias parameters). The inference analysis in [5] is conceptually different than ours; it infers type annotations at a fine level of granularity (i.e., for each variable and expression) and that appears to hinder scalability. Our analysis, which is based on Soot, and the efficient inclusion-based Andersen-style points-to analysis in Spark, appears to scale substantially better, both in terms of time and memory.

Agarwal and Stoller [3] infer ownership types for race-free Java. Their inference algorithm is based on dynamic analysis and thus the inferred types may be unsound. In contrast, our analysis is a safe static analysis and our study indicates that it may be adequately precise.

Recent work by Rayside et al. [36] emphasizes the relevance of ownership inference and visualization. The paper however, appears to be preliminary because it does not present empirical results. Our work uses a related ownership model, but a conceptually different inference analysis. It presents a detailed empirical investigation that indicates that the analyses are practical and adequately precise.

Immutability inference. Porat et al. [34] describe an analysis that detects immutable fields. Their analysis is context-insensitive, libraries are not analyzed and the paper discusses only static fields. Our immutability analysis incorporates limited context sensitivity, analyses large libraries and focuses on instance fields.

Ryder et al. [43] present a framework for side-effect analysis for C that is parameterized by points-to analysis. Our inference analysis uses the same general idea for propagation of side-effects. However, we consider underlying context-insensitive points-to analysis combined with limited context sensitivity during propagation; this combination helps achieve scalable analysis.

Rountev [38], and Salcianu and Rinard [44] present analyses that identify side-effect-free methods in Java programs. In both cases the analyses are applied on relatively small programs (hundreds of reachable methods). Our analysis identifies immutable fields and is applied on substantially larger programs (close to ten thousand reachable methods).

UML class diagrams. Various researchers have studied formalizations of the notions of composition, aggregation and association in the UML [11, 7, 21, 18]. Our work focuses on static analyses that enhance associations, a relatively well-defined concept in the UML.

7. CONCLUSIONS AND FUTURE WORK

We present a mechanism for alias control that is based on the use of ownership and immutability constraints on associations in UML class diagrams. We propose ownership and immutability models and and develop corresponding inference analyses. We perform an empirical study which indicates that the analyses are precise and practical and can support model-driven development and effective reasoning about software quality and software security.

In the future, we plan to perform experiments on more benchmarks; clearly, the results need to be confirmed on a larger code base and by additional studies of absolute precision. Also, we will investigate techniques for further reduction of analysis cost; the cost can be reduced by proving large portions of the library code irrelevant to the ownership and immutability properties. Most importantly, we plan to integrate the analyses into an open-source UML tool.

8. REFERENCES

[1] Ashes suite collection. http://www.sable.mcgill.ca/software.

[2] Dacapo benchmakr suite. http://www-ali.cs.umass.edu/dacapo/gcbm.html.

[3] R. Agarwal and S. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, pages 149–160, 2004.

[4] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[5] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.

[6] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341, 1996.

[7] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J.-M. Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE TSE*, 29(5):459–470, 2003.

[8] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *PLDI*, pages 103–114, 2003.

[9] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, 2004.

[10] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.

[11] J.-M. Bruel, B. Henderson-Sellers, F. Barbier, A. L. Parc, and R. B. France. Improving the UML metamodel to rigorously specify aggregation and composition. In *International Conference on Object-Oriented Information Systems*, pages 5–14, 2001.

[12] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[13] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[14] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment time confinment checing. In *OOPSLA*, pages 374–387, 2003.

[15] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, pages 241–253, 2001.

[16] D. Grove and C. Chambers. Call graph construction in object-oriented languages. *ACM TOPLAS*, 23(6):685–746, Nov. 2001.

[17] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA*, pages 108–124, 1997.

[18] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *OOPSLA*, pages 301–314, 2004.

[19] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.

[20] G. Kniesel and D. Theisen. JAC-access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[21] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *Working Conference on Reverse Engineering*, pages 81–91, 2001.

[22] P. Lam and M. Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *ECOOP*, pages 275–302, 2003.

[23] C. Larman. *Applying UML and Patterns*. Prentice Hall, 2nd edition, 2002.

[24] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.

[25] O. Lhotak and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *CC*, pages 47–64, 2006.

[26] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *PASTE*, pages 73–79, 2001.

[27] A. Milanova. Precise identification of composition relationships for UML class diagrams. In *ASE*, pages 76–85, 2005.

[28] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–12, 2002.

[29] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 14(1):1–42, 2005.

[30] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.

[31] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.

[32] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *In Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, 2002.

[33] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, 1994.

[34] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, 2000.

[35] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.

[36] D. Rayside, L. Mendel, R. Seater, and D. Jackson. An analysis and visualization for revealing object sharing. In *Workshop on Eclipse technology eXchange*, pages 11–15, 2005.

[37] A. Rountev. *Dataflow Analysis of Software Fragments*. PhD thesis, Rutgers University, 2002.

[38] A. Rountev. Precise identification of side-effect free methods. In *ICSM*, pages 82–91, 2004.

[39] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

[40] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java softwhare. *IEEE TSE*, 30(6):372–386, June 2004.

[41] E. Ruf. Effective synchronization removal for Java. In *PLDI*, pages 208–218, 2000.

[42] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2nd edition, 2004.

[43] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, Mar. 2001.

[44] A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[45] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[46] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an emprirical comparison. Technical report, U. Passau, Sept. 2000.

[47] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *OOPSLA*, pages 264–280, 2000.

[48] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

[49] M. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[50] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *CC*, LNCS 1781, pages 18–34, 2000.

[51] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *SAS*, pages 180–195, 2002.

[52] J. Whaley and M. Lam. Cloning-based

context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.

Appendix A

```
class Register {
  private ProductCatalog catalog;
  private Sale sale;
  public Register() {
   catalog=new ProductCatalog(); }
  public void enterItem(Integer id, int q) {
   ProductSpec spec = catalog.getSpec(id);
   sale.makeLineItem(spec, q); }
  public void makeNewSale() {
   sale = new Sale(); }
  public void makePayment(Money cash) {
   sale.makePayment(cash);
   Money balance = sale.getBalance(); }
  public void endSale() {
   sale.becomeComplete(); }
}

class ProductCatalog {
  private Hashtable specs = new Hashtable();
  ProductCatalog() {
   ProductSpec ps;
   ps = new ProductSpec(100,3,"1stItem");
   specs.put(new Integer(100),ps); }
  ProductSpec getSpec(Integer id) {
   return (ProductSpec) specs.get(id); }
}

class Sale {
  private Vector lineItems = new Vector();
  private Payment payment;
  public Money getBalance() {
   return payment.getAmount().minus(getTotal()); }
  public void makeLineItem(ProductSpec s, int q) {
   lineItems.add(new SalesLineItem(s,q)); }
  public Money getTotal() {
   Money total = new Money();
   Iterator i = lineItems.iterator();
   while (i.hasNext()) {
    SaleLineItem sli = (SaleLineItem) i.next();
    total.add(sli.getSubtotal()); }
   return total; }
  public void makePayment(Money cash) {
   payment = new Payment(cash); }
  public void becomeComplete() { //log... }
}

class SaleLineItem {
  private int quantity;
  private ProductSpec spec;
  public SaleLineItem(ProductSpec s, int q) {
   this.spec = s; this.quantity = q; }
  public Money getSubtotal() {
   return spec.getPrice().times(quantity); } BAD!!!
}

public static void main() {
  Register register = new Register();
  while (...more sales...) {
   register.makeNewSale();
   while (...more items...) {
    register.enterItem(new Integer(id),q); }
   register.makePayment(new Money(amount));
   register.endSale(); }
}
```